

Finding Parallelism for Future EPIC Machines

Matthew Iyer[†], Chinmay Ashok[†],
Neil Vachharajani[‡], Joshua Stone^{†*},
Daniel Connors[†], Manish Vachharajani[‡]

[†]University of Colorado, Boulder, CO

[‡]Princeton University, Princeton, NJ

^{*}Now at Intel Corporation, Santa Clara, CA



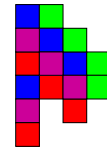
Colorado Computer Architecture Research Group

Parallelism in Current Architectures

```
for(i=0; i<n; i++){
    a[i]=b[i]+c[i]
}
```



Dynamic Schedule



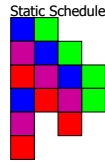
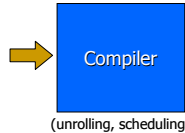
- Traditional models focus on exploiting near ILP in hardware
 - Near ILP: within range of 2K of instructions
 - Speculation and multithreading [Lam '92, Austin '92, Wall '93]
 - Reduced memory latency [Jouppi '97]
 - Register renaming [Wall '93, Postiff '98, Lee '00]
 - Large instruction windows [Austin '92]
- Kilo-instruction machines [Valero '04] proposed



2

Parallelism in Current EPIC Architectures

```
for(i=0; i<n; i++){
    a[i]=b[i]+c[i]
}
```



(unrolling, scheduling)

- EPIC models focus on exploiting near-ILP using software
 - Larger compiler optimization and analysis scope
 - Inlining, loop-unrolling, etc.
 - Less able to exploit dynamic properties of execution
 - e.g., variable memory latency [Sias ISCA '04]
 - Difficult to tolerate cache miss penalties



3

Amount of Parallelism

- Several limit studies
 - Flynn's bottleneck of instructional level parallelism [Flynn '70]
 - Control flow limits parallelism [Lam '92]
 - Comprehensive study [Wall '93] shows how various factors affect ILP at the microarchitectural level
 - ILP limits in response to increasing memory latency and branch misprediction penalty [Jouppi '97]
 - Stack aliasing : unnecessary true data-dependences in stack pointer manipulation [Postiff '98, Lam '92]
 - Limitations of procedure and loop boundaries on ILP of an EPIC architecture [Lee '00]



4

Characterizing Parallelism

- Ever-changing hardware and software conditions
 - Workloads and programming concepts
 - Compiler technology (analysis and optimization)
- Prior studies focus on measurement and lack deep analysis to guide research
- Ideal parallelism: upper bound on the inherent parallelism in the implementation of an algorithm
 - Programming model, compiler, and ISA are involved
 - Examine complete program execution
- Studies must be **comparative** to allow conclusions that **characterize the sources of ILP**



5

Distant Parallelism

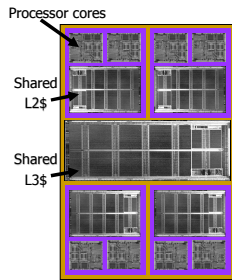
- Eliminate barriers to observing parallelism
 - Stack activity
 - Control flow
 - Resources
- Study different effects of ISAs and compilers
- Discovery: critical dependences release many distant parallel instructions
 - Independent execution regions exist and form local dependence chains

```
cfg_t * cfg;
Func_t * func;
Program_t * prog;
...
prog = read(binary);
...
while(func=prog->first;
      func!=prog->last;
      func=func->next) {
    cfg=init(func);
    dom_analysis(cfg);
    pdom_analysis(cfg);
    live_analysis(cfg);
    optimize(cfg);
}
```



6

Extracting Distant-ILP for Emerging Machines



- All chips will have multiple processor cores
 - Chip multiprocessor (CMP) and chip multi-threaded (CMT)
 - Potentially multithreaded (SMT/MT) in each core
 - Lower communication costs
 - Purpose-built low-latency communication hardware
 - Inexpensive speculation (e.g. transactional memory)
- Challenges
 - Developing effective parallel programming models
 - Extracting parallelism from single-thread application for these machines



7

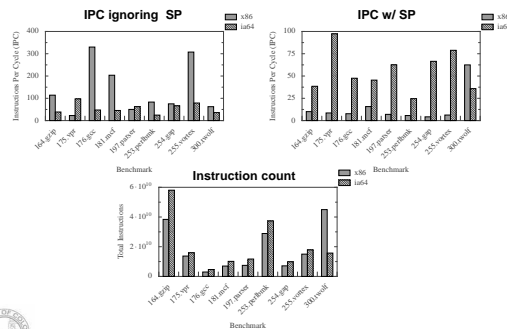
Methodology

- Gather execution traces for SPEC-CPU2000-INT
 - Optimized binaries (-O3) compiled using gcc and icc
 - Pin tool [Luk '05] dynamically instruments benchmark binaries to collect trace information
 - Traces are compressed with TCgen [Burtcher '04]
 - Architectures: Itanium, x86, PowerPC
- Schedule traces on an "ideal" machine using Adamantium
 - Only models true data dependences
 - Eliminates crippling dependences such as the stack pointer
 - Performs perfect branch prediction
 - Executes all instructions in a single cycle
 - Instruction window size can be varied



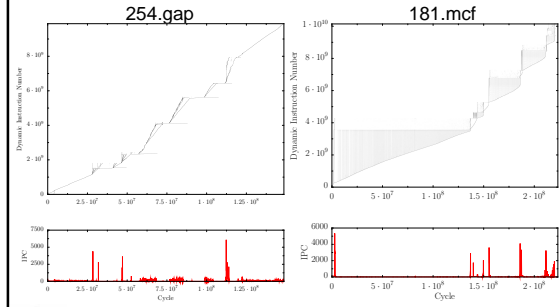
8

IPC Overview



9

Amount of Distant ILP

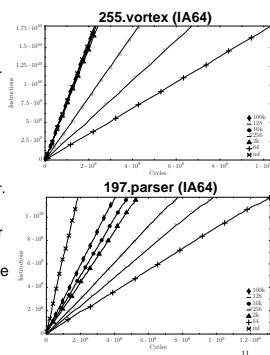


Top Graph: Vertical lines indicate distant parallelism

10

Effect of Instruction Window Size

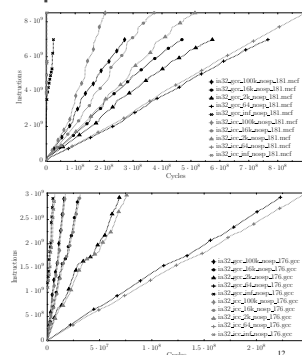
- 255.vortex (top) maxes out at a 2k instruction window.
 - 300.twolf also had this behavior.
- 197.parser (bottom) exhibits unbounded increases in parallelism.
 - The remaining benchmarks we studied showed similar behavior.
- We found that ILP limitations for larger windows in benchmarks such as 255.vortex on IA64 were due to predicate dependences.



11

Effect of the Compiler on ILP

- More aggressive compiler expected to perform better.
- Compilers are designed to exploit nearby ILP.
- Better compiler might hurt its own chances of exploiting distant parallelism.



12

Compiler effect comparison

- For x86 - icc has a lower average IPC but is faster by executing fewer instructions.
- For Itanium - neither compiler seems to have an advantage.

The figure consists of four bar charts arranged in a 2x2 grid, comparing the performance of two compilers, **icc** (Intel C++ Compiler) and **gcc** (GNU Compiler Collection), across various benchmarks.

Top Row: x86 Architecture

- Left Chart (x86 - icc):** Shows **Instructions per Cycle (IPC)** on the y-axis (0 to 1000) for benchmarks: 100, 175, 1000, 10000, 25000, and 50000. The **icc** compiler (dark bars) consistently shows higher IPC values than **gcc** (light bars) across all benchmarks, with a significant peak for the 1000 benchmark.
- Right Chart (x86 - gcc):** Shows **Instructions per Cycle (IPC)** on the y-axis (0 to 100) for the same benchmarks. The **gcc** compiler (light bars) consistently shows higher IPC values than **icc** (dark bars) across all benchmarks.

Bottom Row: Itanium Architecture

- Left Chart (Itanium - icc):** Shows **Time to execution** on the y-axis (0 to 40) for benchmarks: 100, 175, 1000, 10000, 25000, and 50000. The **icc** compiler (dark bars) generally shows lower time to execution (faster) than **gcc** (light bars) for most benchmarks, particularly for 1000 and 10000.
- Right Chart (Itanium - gcc):** Shows **Time to execution** on the y-axis (0 to 75) for the same benchmarks. The **gcc** compiler (light bars) generally shows lower time to execution (faster) than **icc** (dark bars) for most benchmarks, particularly for 100 and 175.

Legend: **icc** (dark bars), **gcc** (light bars)

Comparing Architectures (IA64 vs. PPC)

The figure consists of four plots arranged in a 2x2 grid. The top row shows 'Dynamic Instruction Counts' for '181.mcf IA64' (left) and '181.mcf PPC' (right). The y-axis for these plots ranges from 0.00 to 1.75 x 10⁹. The x-axis for the IA64 plot ranges from 0 to 1.25 x 10⁸, while for the PPC plot it ranges from 0 to 6 x 10⁸. Both plots show a shaded area representing a range of counts, with a line indicating a specific trend. The bottom row shows 'IPC' (Instructions Per Cycle) versus 'Cycle' for IA64 (left) and PPC (right). The y-axis for these plots ranges from 0 to 2 x 10⁹. The x-axis for the IA64 plot ranges from 0 to 1.25 x 10⁸, while for the PPC plot it ranges from 0 to 6 x 10⁸. Both plots show a red line with several sharp peaks, indicating high IPC values at specific cycles.

- PPC with stack-pointer dependence versus Itanium with predicate register dependences

14

Comparing Architectures (IA64 vs. x86)

The figure consists of four plots arranged in a 2x2 grid, comparing IA64 and x86 architectures for the 181.mcf benchmark.

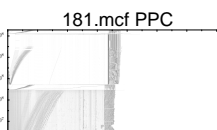
- Top Left Plot (IA64):** Dynamic Instruction Scales (Y-axis, 0.00 to 1.75 $\times 10^9$) vs. Cycles (X-axis, 0 to 7 $\times 10^9$). The plot shows a large shaded region representing the execution space, with a significant portion of the area shaded in gray, indicating high instruction scales.
- Top Right Plot (x86):** Dynamic Instruction Scales (Y-axis, 0.00 to 1.50 $\times 10^9$) vs. Cycles (X-axis, 0 to 1 $\times 10^9$). The plot shows a large shaded region representing the execution space, with a significant portion of the area shaded in gray, indicating high instruction scales.
- Bottom Left Plot (IA64):** IPC (Y-axis, 0 to 1.5) vs. Cycles (X-axis, 0 to 7 $\times 10^9$). The plot shows a series of sharp peaks, indicating high IPC values at specific points in the execution.
- Bottom Right Plot (x86):** IPC (Y-axis, 0 to 2) vs. Cycles (X-axis, 0 to 1 $\times 10^9$). The plot shows a series of sharp peaks, indicating high IPC values at specific points in the execution.

■ IA64 without predicate register dependences versus x86 ignoring stack-pointer manipulations


15

Comparing Architectures (PPC vs. x86)


181.mcf PPC



181.mcf x86



- Chain of dependences at end of PPC
- Similar dependence chain in middle of x86



16

Independent Distant ILP on x86

253.perlbnk

164.gzip


The figure displays four plots arranged in a 2x2 grid, comparing the performance of two benchmarks, 253.perlbnk and 164.gzip, under Independent Distant ILP on x86. The top row shows the Dynamic Instruction Number (Y-axis, ranging from 0.0 to 2.5 $\times 10^9$) versus Cycle (X-axis, ranging from 0 to 2 $\times 10^8$). The bottom row shows the Instruction Cache (I/C) hit/miss rate (Y-axis, ranging from 0 to 4000) versus Cycle (X-axis, ranging from 0 to 2 $\times 10^8$).

The top-left plot (253.perlbnk) shows multiple execution traces (gray lines) that generally increase linearly, indicating a steady rate of instruction execution. The top-right plot (164.gzip) shows multiple execution traces (gray lines) that exhibit significant fluctuations, with some traces showing a sharp increase in instruction number around cycle 1 $\times 10^8$.

The bottom-left plot (253.perlbnk) shows the I/C hit/miss rate (red line) which remains low (near 0) throughout the execution, indicating a high hit rate. The bottom-right plot (164.gzip) shows the I/C hit/miss rate (red line) which exhibits several sharp peaks, reaching up to 4000, indicating a high miss rate during those periods.

Conclusions

- ILP differs based on ISA
 - When stack-pointer (x86 and PPC) and predicate dependences (IA64) are ignored, the overall distant ILP pattern is similar
- Distant ILP manifests in application specific ways
 - Critical dependences (dependences that uncover large amounts of parallelism)
 - Independent chains of localized dependences
- Compiler helps and hinders
 - Techniques to improve nearby ILP can harm distant ILP
- Future work
 - Critical dependences - can value prediction be used?
 - Independent chains - can separate threads be spawned?
 - Add realistic branch prediction and memory latency models
 - How much distant ILP can be harnessed in emerging CMP systems?
 - Resolve remaining differences among ISAs



18